Indexing Compressed Text

Edleno S. de Moura[1] Gonzalo Navarro[23] Nivio Ziviani[13]

[1] Depto. de Ciência da Computação, Universidade Federal de Minas Gerais, Brazil.
[2] Depto. de Ciencias de la Computación, Universidad de Chile, Chile.
[3] This work has been partially supported by Brazilian CNPq Project 520916/94-8, RITOS/CYTED Project, and Chilean Fondecyt grant 1-950622.

Abstract. We present a technique to build an index based on suffix arrays for compressed texts. We also propose a compression scheme for textual databases based on words that generates a compression code that preserves the lexicographical ordering of the text words. As a consequence it permits the sorting of the compressed strings to generate the suffix array without decompressing. As the compressed text is under 30% of the size of the original text we are able to build the suffix array twice as fast on the compressed text. The compressed text plus index is 55-60% of the size of the original text plus index and search times are reduced to approximately half the time. We also present analytical and experimental results for different variations of the word-oriented compression paradigm.

1 Introduction

The amount of textual information available worldwide is experimenting an impressive growth in the last years. The widespread use of digital libraries, office automation systems and document databases are some examples of the kind of requirements that are becoming commonplace. Phenomena like the World Wide Web and indexing mechanisms over the Internet definitely feed this explosion of textual information electronically available. Therefore, compression appears always as an attractive choice, if not mandatory. However, the combination of text compression and the retrieval requirements of textual databases does not always succeed. Because of this, many textual databases schemes do not compress the text, while compression is left to data that is not to be queried.

Approaches to combine text compression and indexing techniques using inverted lists have recently received some attention [MB95, WBN92, ZM95]. However, work on combining compression techniques and suffix arrays has not been pursued. Suffix arrays [MM90] or Pat arrays [Gon87, GBYS92] are indexing structures that achieve space and time complexity similar to inverted lists. Their main drawback is their costly construction and maintenance procedure. However, suffix arrays are superior to inverted lists for searching long phrases or complex queries like regular expression searching, longest repetitions and most frequent searching [GBYS92, MM90].

In this work we describe an efficient compression scheme to be used in conjunction with suffix arrays. We make three main contributions. First, we study

analytically and experimentally different variations of the word-oriented compression paradigm [BSTW86]. Second, we define an encoding method that preserves the lexicographical ordering of the text words. This idea already existed from a long time ago [Knu73], but to the best of our knowledge, it had not been applied in practice. Third, based on our encoding method, we describe a mechanism to build suffix arrays for either compressed or uncompressed texts which is nearly twice as fast when compared with the standard suffix array construction. For compressed texts querying times are reduced to approximately half the time. It is typical that the compressed text takes less than 30% of its original size, and the whole index (text and suffix array) is reduced to 60%, which is less than the space of the uncompressed text with no index.

Another type of text compression scheme is proposed in [Man93]. The main purpose of [Man93] is to speed up sequential searching by compressing the search key rather than decompressing the text being searched. As a consequence it requires no modification in the algorithm and any known linear searching algorithm can be used. For typical text files the compressed text in [Man93] is about 70% of its original size, while in our scheme is below 30% of its original size. The works by [FT95, ABF96] also permit sequential search directly in the Lempel-Ziv compressed text [ZL77, ZL78]. However, we deal with the problem of implementing random access into a compressed text while their main purpose is to search sequentially the compressed file.

For the experimental results we used literary texts from the 2 gigabytes TREC collection [Har95]. We have chosen the following texts: AP Newswire (1989), DOE - Short abstracts from DOE publications, FR - Federal Register (1989), WSJ - Wall Street Journal (1987, 1988, 1989) and ZIFF - articles from *Computer Selected* disks (Ziff-Davis Publishing). Table 1 presents some statistics about the five text files. We considered a word as a contiguous string of characters in the set {A...Z, a...z, 0...9} separated by other characters not in the set {A...Z, a...z, 0...9}.

Files	Text		Vocabulary		Vocab./Text	
	Size (bytes)	#Words	Size (bytes)	#Words	Size	#Words
AP	237,766,005	38,977,670	1,564,050	209,272	0.65%	0.53%
DOE	181,871,525	28,505,125	1,949,140	235,133	1.07%	0.82%
FR	219,987,476	34,455,982	1,284,092	181,965	0.58%	0.52%
WSJ	262,757,554	42,710,250	1,549,131	208,005	0.59%	0.48%
ZIFF	242,660,178	39,675,248	1,826,349	255,107	0.75%	0.64%

Table 1. Text files from the TREC collection

This paper is organized as follows. In Section 2 we study different alternatives to word-based compression. In Section 3 we show how to encode while preserving lexicographical order. In Section 4 we present the indexing technique. Finally, in Section 5 we present some conclusions and future work directions.

2 Compression Method

In this section we describe and justify the compression method used, which is composed of a semi-static model and Huffman encoding. We also support our arguments with analytical and experimental data.

2.1 Semi-Static Models and Text Compression

Assumptions that are normally made when designing a general compression scheme are not valid for textual databases. For example, the need of direct access to parts of the text immediately rules out *adaptive* models, which are pervasive in modern compression schemes [ZM95].

Adaptive models start with no information about the text and progressively learn about its statistical distribution as the compression process goes on. They are one-pass and store no additional information apart from the compressed data. In the long term, they converge to the true statistical distribution of the text. On the other hand, they cannot start decompressing a file except from the beginning, since they need to learn the same statistical data while decompressing.

It could be argued that a large text could be divided into smaller parts that are compressed by an adaptive algorithm, thus allowing direct access to each part. However, as [ZM95] points out, adaptive methods need to process some data before learning the distribution and making compression effective. To obtain a reasonable compression ratio, this amount of data must be at least 10 megabytes in practice, which is prohibitively large for direct access.

If adaptive methods are ruled out, we can still avoid storing additional information on the compressed data by using a *static* model. In that case, an average distribution is assumed for the database. However, these models are not flexible and tend to achieve poor compression ratios when the data deviates from their statistical assumptions. It is very difficult to find a static model that presents a good compression rate for the wide range of text distributions.

We are left with *semi-static* models. These do not assume any distribution on the data, but learn it in a first pass. In a second pass, they compress the data by using a fixed code derived from the distribution learned. The disadvantages are that they must make two passes and that the encoding data must be stored to decompress. However, they have an advantage which is crucial for our purposes: since the same code is used at every point in the data, direct access is possible.

The disadvantages exposed against adaptive models are not very important in our paradigm. The need for two passes on compression at indexing time is compensated by faster decompression at query time, which tends to be much more frequent than indexing. The need to store the distribution of the text is not a new requirement, since an index for the text has to be kept anyway. The decompression information poses a negligible overhead over normal index space.

Another central requirement on a compressed database is decompression efficiency. Decompression is faster on semi-static methods than on adaptive methods because the last ones need to update the statistical information as they decompress.

2.2 Compressing Words

The most used encoding methods are arithmetic coding [WNC87], Ziv-Lempel coding family [ZL77, ZL78] and Huffman coding [Huf52]. Arithmetic coding can achieve better compression ratios than Huffman coding because of its ability to use fractional parts of bits. However, as in the arithmetic coding the data is encoded as rational numbers there is no way of knowing exactly where a given word starts in the compressed data. This is inherent to the coding method, which means that any compression method that uses arithmetic coding presents this kind of difficulty, independently of the kind of model being used (static, semi-static or dynamic). A similar problem occurs with Ziv-Lempel codings, as they can be used only with adaptive modeling, which make both arithmetic and Ziv-Lempel codings unfeasible to use with indexing. On the other hand, the unability to use fractional parts of bits prevents Huffman coding to reach the optimal compression ratio. However, it has been shown that when using a word-based model on natural language text, the compression inefficiency of Huffman is no more than 9% over arithmetic coding [ZM95]. We show later on analytical and experimental evidence supporting this.

A successful idea to merge compression and textual databases is to take words instead of characters as symbols [BSTW86, Mof89, BMN+93]. This is because words are the atoms on which most information retrieval systems are built. Moreover, much better compression ratios are achieved by taking words as symbols than taking characters. The reason behind this fact is that words carry a lot of meaning in natural language, their distribution having much more to do with the semantic structure of the text than their individual letters. Of course, similar results can be achieved by using a frequency model of higher order on characters (i.e. where the distribution is a function of the previous characters read), but words have the further advantage that their number is not as large as the number of character sequences of length, say, five.

Since the text is not only composed of words but also of separators, a model must also be chosen for them. An obvious possibility is to consider the different inter-word separators as symbols too, and make a unique alphabet for words and separators. However, this idea is not using a fundamental *alternation* property: words and separators always follow one another. In [Mof89, BMN+93] two different alphabets are used: one for words and one for separators. Once it is known that the text starts with word or separator, there is no confusion on which alphabet to use. We call this model *separate alphabets*.

A semi-static model and Huffman coding [Huf52] form a good compression method for texts [ZM95, BMN+93]. Moreover, we show later that this combination allows very interesting variations. Figure 1 presents an example of compression using Huffman coding on words for separate alphabets. The set of symbols for the alphabet of words is {`a, each, is, rose`}, whose frequencies are 2, 1, 1 and 3, respectively, and the set of separators is {",␣", "␣"}, whose frequencies are 1 and 5, respectively (␣ represents a space). to more frequent symbols. Given n symbols and their zero-order frequencies in the text, the algorithm builds the optimal trie in $O(n \log n)$ time.

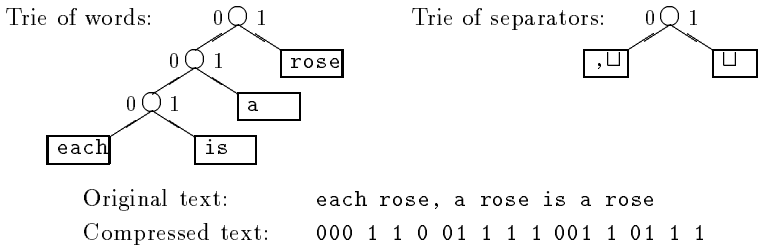

Fig. 1. Compression using Huffman coding for separate alphabets

2.3 Compression Based on Spaceless Words

We study now a different alternative to use the structure of words and separators. The main idea is to consider the separator that follows a word as part of the same word. In the example above, `rose␣` and `rose,␣` are different words. In this scheme, the separators do not exist anymore independently. Although it may seem that we are saving the bits to encode separators, this is not the case. Making words different because of being followed by different separators completely changes their statistical distribution, loosing a lot of matching with the semantics of the text. Moreover, the number of different "symbols" increase drastically, what forces to use more bits to differentiate them. Figure 2 shows this phenomenon for the WSJ text file.

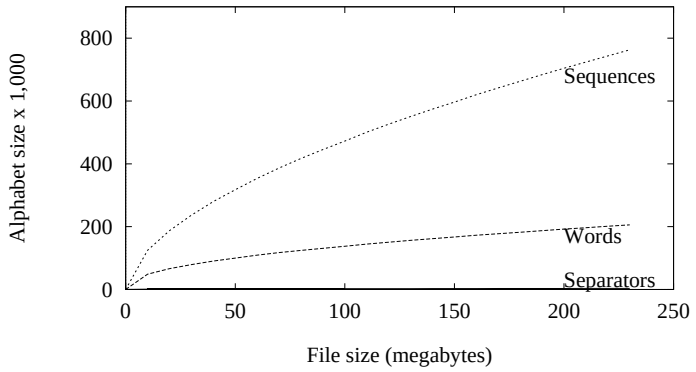

Fig. 2. Alphabet size for separators, words and sequences for the WSJ file

We propose a new variation for the separate alphabets model, which we call *spaceless words*. An important fact that is not used in the method of separate alphabets is that a word is followed by a single space in most cases, as shown in Table 2. We then take this as a *default*. That is, if a word is followed by a space, we just encode the word. If not, we encode the word and then the separator. At decoding time, we decode a word and assume that a space follows, except if the next symbol corresponds to a separator. Of course the alternation property

does not hold anymore, so we have a single alphabet for words and separators (single space excluded).

Files	Words (n)	Single spaces (s)	s/n
AP	38,977,670	30,020,803	0.77
DOE	28,505,125	22,001,366	0.77
FR	34,455,982	25,506,763	0.74
WSJ	42,710,250	33,909,610	0.79
ZIFF	39,675,248	27,225,864	0.69

Table 2. Number of single-space separators for the TREC collection

Figure 3 presents the original and compressed texts for the same example presented previously using Huffman coding for spaceless words.

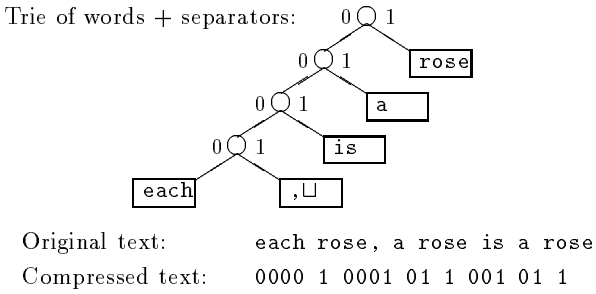

Fig. 3. Compression using Huffman coding for spaceless words

We show later experiments supporting the thesis that the new variation we are proposing achieves slight better compression ratios.

2.4 Analytical and Experimental Results

We first analyze the zero-order entropy of natural language text when words are symbols, using Zipf's law [Zip49]. We then present experimental results showing that Huffman coding is very close to the entropy limit, thus supporting the thesis that it is a good choice for our purposes. We also compare the separate alphabets, spaceless words and character-based models. We use N to denote the number of characters in the text and n for the size of the vocabulary (number of different words).

Zipf's law states that, if we order the words of a natural language text in descending order of probability then the probability of the first word is i times the probability of the i-th word, for every i. This means that the probability of the i-th word is $p_i = 1/(iH_n)$, where $H_n = \sum_{j=1}^{n} 1/j \ = \ln n + \gamma + O(1/n)$ and $\gamma \approx 0.57$ is the Euler's constant.

We analyze the binary entropy of such distribution for a vocabulary of n words, as follows.

$$\sum_{i=1}^{n} p_i \log_2 \frac{1}{p_i} \quad = \quad \frac{1}{\ln 2} \sum_{i=1}^{n} \frac{\ln H_n + \ln i}{i H_n}$$

$$= \frac{1}{H_n \ln 2} \left(\ln H_n \sum_{i=1}^{n} \frac{1}{i} + \sum_{i=1}^{n} \frac{\ln i}{i} \right) \quad = \quad \log_2 H_n + \frac{1}{H_n \ln 2} \sum_{i=1}^{n} \frac{\ln i}{i}$$

and by observing that

$$\sum_{i=1}^{n} \frac{\ln i}{i} \quad \geq \quad \int_2^{n+1} \frac{\ln x}{x}\, dx \quad = \quad \frac{\ln(n)^2 - \ln(2)^2}{2} + O(1/n)$$

(the bound is tight, since an upper bound is $\ln(n)^2/2$), we have that the entropy is at least

$$\frac{1}{2} \log_2 n + \log_2 \ln n - \frac{\gamma}{2 \ln 2} + O\left(\frac{1}{\log n}\right)$$

A first observation is that the entropy is approximately half of the length of a naive word encoding (which is of course $\log_2 n$). This shows that using compression on words instead of simply taking words as symbols allows to save 50% more space. As explained, the good point is that we can achieve the compression of a near 6th order source with much less storage requirements.

A second observation comes from applying the formula to real cases. For a vocabulary of size $n = 2^{18}$ (a typical value in our texts), the formula predicts that each word will take near 13 bits on average on the best compression scheme that takes words separately. Dividing 13 by the average word length (6 characters, or 48 bits) we get 27% of the original size. With an alphabet of $n = 2^{17}$ (also typical) we get 24%. Table 3 shows that Huffman coding on words is approximately 2% worse than the entropy. This shows that the method is efficient and has very small compression overhead. The table also shows that spaceless words is marginally better than separate alphabets, and that both are much better than character-based compression.

Files	Separate Alphabets		Spaceless Words		Character Based		Commercial Programs	
	Huff	Entrp	Huff	Entrp	Huff	Entrp	Gzip	Compress
AP	27.7%	26.1%	27.3%	26.2%	57.8%	57.5%	38.6%	43.8%
DOE	26.7%	24.4%	26.4%	24.6%	57.2%	57.0%	34.8%	41.0%
FR	27.0%	25.1%	26.9%	25.3%	59.4%	59.1%	27.8%	38.6%
WSJ	28.3%	25.9%	27.1%	26.0%	58.5%	58.0%	37.5%	43.0%
ZIFF	28.9%	27.2%	28.9%	27.5%	61.3%	60.9%	34.1%	41.6%

Table 3. Compression ratio and entropy for different models.

3 Huffman Coding that Preserves Lexicographical Order

As we will see later on, when building an index based on a suffix array, a large number of pairs of strings from the text have to be compared. If the text is already compressed, the need to decompress for each comparison significantly slows down the indexing process. An obvious solution is to build the index before compressing, later remapping the pointers to the compressed text. In this section we present a better solution that works directly on the compressed file, performing the comparisons without decompressing the text at all. This leads to a speedup over the normal process because less comparison operations are made as the codes are shorter than the strings. Moreover, buffering is improved since more compressed text fits into the same buffer space.

The idea is to redefine the conditions for Huffman coding. In the original Huffman algorithm the expected length of an encoded word is minimized, and the algorithm builds a trie that minimizes that length. We redefine this constraint by forcing the obtained code to keep the lexicographical ordering over the words (symbols). This is equivalent to requiring that a left-to-right pass over the leaves of the Huffman trie obtains the words of the vocabulary in lexicographical order. The problem now is to obtain the minimum expected-length code that preserves that order among words.

If that property holds, we do not need to decompress the different text positions to compare, since the result is the same if the compressed codes are compared. Moreover, we do not need to compare the codes bit by bit. Instead, we compare complete computer words, which makes the process much faster on most architectures. A couple of points on comparing complete computer words deserve careful explanation. A first point is that, by comparing complete words, we compare trailing bits that are outside our interest. However, since in a Huffman coding no code is a prefix of another, we can guarantee that if the first word of two strings differ, their codes will differ before they end, i.e. before the trailing bits have a chance to influence the comparison. If the first words of each string are equal, their codes are equal too and hence the comparison proceeds correctly with the code bits of the second words, and so on. This is true because, as we show later, only words (not separators) are represented in the text that is indexed. Further, since we index only word beginnings, all codes also represent beginnings of complete words.

A second point regards efficiency. Take the normal case of a 32-bit architecture. The codes of the strings do not necessarily begin in a computer word boundary. This means that to obtain each 32-bit group of a compressed word, we must perform some *shift* and *or* bit operations. More specifically, we need 5 operations the first time and 3 at each subsequent 32-bit group. This slows down the comparison process if the original strings differ in their first characters. On the other hand, since we typically pack a word in 12-13 bits, we are comparing 17-18 uncompressed characters in the first computer word, at a cost of 5 operations. This is normally enough to distinguish two different strings of the text. Hence, the comparison is normally very fast. Further, if the strings are equal in their first 5 characters, comparing compressed words is faster (and can be much

faster if the strings share a long prefix). Observe that sharing long prefixes is a phenomenon that becomes more common as the sorting process progresses.

We still do not specify how to build a minimum expected length encoding that preserves lexicographical ordering. The algorithm to build this code is in fact known, although it comes from a different area. Imagine we have n keys $k_1, k_2, \ldots, k_n$, whose access probabilities are $p_1, p_2, \ldots, p_n$, respectively, and we want to build a minimum average cost binary search tree having the keys at the leaves. It is not difficult to see that the resulting tree has to minimize

$$\sum_{i=1}^{n} (\text{depth of leaf } i) \times p_i$$

since the depth of a key is the cost to search it. Moreover, the tree cannot alter the order of the keys, since it is a binary search tree. This is also the optimal trie we want, since we minimize the same expression (the depth is the length of the code for each leaf).

The solution to this problem is presented by Hu and Tucker [HT71] and also considered in [Knu73], where the tree is built with an $O(n \log n)$ algorithm. Therefore, the complexity is the same as for obtaining a Huffman code.

A natural question is how far is the resulting code from the optimal classical Huffman code. In [Knu73] it is shown that the difference between the average length of a code under both methods cannot be larger than 2 bits. That means an increment of only 4-5% in the compression ratios (with respect to the original text), which is more than reasonable compared to the indexing speed achieved.

Table 4 shows experimental results comparing the compression ratios of both Huffman and Hu-Tucker codings using separate alphabets and spaceless words models. Note that Hu-Tucker coding using the spaceless words model presents better compression ratios than for the separate alphabets model, results that are only 0.5% worse than the Huffman coding.

Files	Separate Alphabets		Spaceless Words	
	Huffman	Hu-Tucker	Huffman	Hu-Tucker
AP	27.71%	29.47%	27.31%	27.75%
DOE	26.70%	28.17%	26.40%	26.53%
FR	27.01%	28.43%	26.87%	27.06%
WSJ	28.28%	29.30%	27.13%	27.40%
ZIFF	28.86%	30.32%	28.93%	29.29%

Table 4. Compression ratio for the different codings and models

Table 5 shows the time necessary to compress the texts from the TREC collection for Huffman and Hu-Tucker codings using separate alphabets and spaceless words models, and two known commercial software, Gzip and Compress. Compression times for Hu-Tucker and Huffman algorithms are approximately the

same (Hu-Tucker is more complex but we use canonical Huffman trees[4], which allow faster decompression but have slightly more costly construction). As we show later, the gains in *indexing* times are much more important. These gains are due to our ability to index compressed text directly if Hu-Tucker compression is used, because it preserves the lexicographical ordering of the codes.

Files	Separate Alphabets		Spaceless Words		Commercial Programs	
	Huffman	Hu-Tucker	Huffman	Hu-Tucker	Gzip	Compress
AP	25.53	23.46	24.00	24.01	28.00	9.66
DOE	18.85	17.28	18.11	18.31	19.30	7.00
FR	21.56	19.75	21.08	21.11	20.98	8.25
WSJ	28.76	25.01	27.85	26.83	29.16	10.48
ZIFF	26.86	23.83	27.28	25.80	26.00	9.93

Table 5. Compression time for different algorithms (in elapsed minutes)

4 Index Construction and Querying

An interesting approach for full-text databases regards the text as a long string [Knu73, MM90, GBYS92]. Each position in the text is called a *suffix*. A suffix is defined by a starting text position and extends to the end of the text. Depending on the character sequences that are to be searched, the user must define the *index points*, i.e. which positions of the text will be indexed. In text databases it is customary to index only word beginnings, and to *filter* the text to be indexed. The filter may map characters (e.g. converting all to lower-case), eliminate stopwords (e.g. `a`, `the`, etc.), disregard blanks, replace synonyms, perform stemming, among others [FBY92]. In the following we assume that the filtering process makes the text case-insensitive and disregards blanks. Of course filtering is used only for indexing purposes, while the original text is unchanged.

A *suffix array* is an array of pointers to suffixes, which is lexicographically sorted by the referenced suffixes. That is, it provides sorted accesses to all suffixes of interest in the text. With a suffix array it is possible to obtain all the occurrences of a string in a text in logarithmic time using binary search. The binary search is indirect since it is necessary to access both the index and the text to compare a search key with a suffix. Figure 4 illustrates the suffix of a text example with seven index points. Each index point corresponds to the address of a suffix. Building a suffix array is similar to sorting variable length records, at a cost of $O(n \log n)$ accesses on average. The extra space used by a suffix array is only one pointer per index point.

[4] These are Huffman trees where the right subtree of every node is not taller than the left subtree. Since the Huffman code is symmetric with respect to left and right trees, exchanging the children to achieve this property is possible. This is not the case in Hu-Tucker coding.

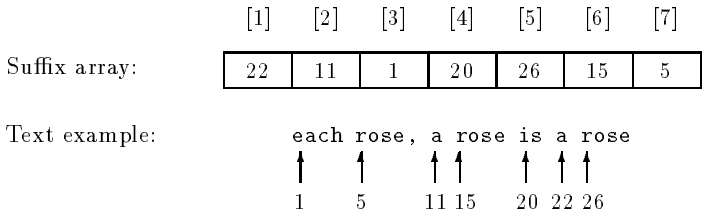

Fig. 4. A suffix array

4.1 Building Suffix Arrays on Compressed Texts

The construction of a suffix array is simply an indirect sort of the index pointers to every suffix of interest in the text. The difficult part is to do the sorting efficiently when large texts are involved. Large texts do not fit in main memory and an external sort procedure has to be used.

Our indexing algorithm is based on the algorithm proposed in [GBYS92] for generating large suffix arrays. The algorithm divides the text in blocks small enough to be individually indexed in main memory. It works with each block separately in three distinct phases. In the first phase it generates the suffix array for the current block located in main memory using internal sorting. In the second phase it reads sequentially from disk all the blocks of text previously processed and generates counters indicating the number of suffixes that fit between each pair of suffixes in the current block (this can be done by performing a binary search in the current block in main memory using the suffix array just generated for the current block). Finally, the suffix array already obtained up to that moment is read sequentially from disk and merged with the current suffix array in main memory guided by the counters obtained in the second phase. These steps are repeated until no more blocks of text are left to index.

In our algorithm we scan the text and obtain the Hu-Tucker encoding of the text soon after filtering the text (in our experiments the filtering process considers any text separator as a unique symbol and upper case letters are converted to lowercase letters). The process of building the suffix array for the original text is carried out on this filtered Hu-Tucker compressed text. Before, an unsorted array pointing to every position that starts a word in the original text is obtained simultaneously with the filtering phase. Then this array is indirectly sorted by comparing the corresponding Hu-Tucker codes.

This sorting can be done with no additional memory requirements or disk space. To accomplish this we adjust the memory utilization during the three phases of the algorithm proposed in [GBYS92]. In the first phase of the algorithm we use the space allocated for the counters (which is not necessary in this phase) to accomodate the unsorted suffix array for the original text together with the filtered suffix array. Both arrays are simultaneously sorted in this phase, guided by the filtered Hu-Tucker compressed text. In the second phase of the algorithm the suffix array for the original text is stored in disk to open space for the

counters. In the third phase (merging phase) the suffix array is read from disk into the space used for the filtered array. At the end the merging output is a suffix array ordered by the Hu-Tucker filtered text, but with pointers to the original text.

Notice that the original text may be uncompressed or even compressed with *another* mechanism (e.g. Huffman, which gives better compression ratios). When compressed text is used it must allow to retrieve the original text, separators included, since that original text would be otherwise lost. As we see later, keeping the original text compressed improves also *query* times.

Our scheme works for any kind of text and the filtered Hu-Tucker compressed text is used to obtain indexing efficiency, independently of the original text to be indexed being compressed or not. If the text was compressed with Huffman, for instance, the words must be decompressed to filter and re-compress with Hu-Tucker, but the final suffix array must point to bits of the Huffman-compressed text. Figure 5 shows the suffix array for a compressed text using Huffman coding and spaceless words model (recall Figure 2).

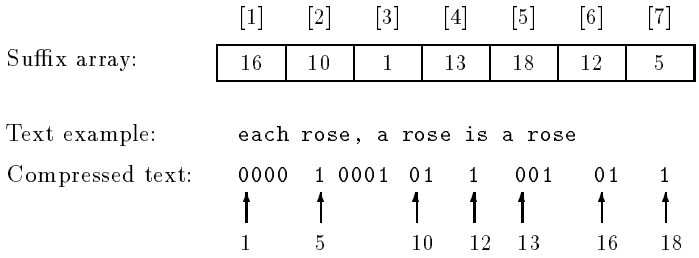
[1] [2] [3] [4] [5] [6] [7]
Suffix array:
16 10 1 13 18 12 5
Text example:
each rose, a rose is a rose
Compressed text:
0000 1 0001 01 1 001 01 1
1 5 10 12 13 16 18

Fig. 5. Suffix array on compressed text

Searching is performed in the usual way: binary searching the pattern in the original text. If the original text is compressed we have to decompress $O(\log N)$ text positions to drive the binary search. This overhead is minimal compared to reduced seek times, as we see in the next section.

4.2 Analytical and Experimental Results

We analyze now the performance of this indexing scheme. Filtering and compression takes $O(N)$ time, which is negligible compared to suffix array construction. The Hu-Tucker algorithm is $O(n \log n)$. As shown in [Hea78], $n = O(N^\beta)$ for $0 < \beta < 1$. For example, for the DOE collection we have $n = 9.43\ N^{0.53}$, while for ZIFF we have $n = 10.77\ N^{0.51}$. Therefore, Hu-Tucker time is $O(N^\beta \log N)$ time, which is dominated by the traversal time. The space requirement is $O(N^\beta)$.

We build the suffix array with the algorithm presented in [GBYS92]. Given M bytes of main memory, the algorithm performs $O((N/M)^2)$ sequential disk readings and takes $O(N^2 \log M/M)$ total CPU time. In the merge phases of the algorithm, main memory is divided in three parts and contains: (a) a text block,

(b) its suffix array taking one computer word per index point of the text, (c) an array of *counters* taking also one computer word per index point. We assume a 32-bit architecture, i.e. pointers and counters of 4 bytes.

Each word contains approximately 6 letters on average. Therefore, in the uncompressed scheme, $6+4+4=14$ bytes are needed to hold each index point. In the compressed scheme, the text is reduced to a 30% at least, therefore we need $1.8+4+4=9.8$ bytes, i.e. 0.7 times the space of the uncompressed version. This effectively multiplies the size of the buffer by 1.43, thus dividing by that proportion the CPU time to index. However, the disk costs dominate in practice, what makes the improvement to be near $1.43^2 \approx 2$. That is, indexing is twice as fast on the compressed text.

Table 6 presents indexing times for the five texts of the TREC collection, under the filtering conditions presented above. We show separately the times to filter/compress and to to obtain the final index. We also show the times taken by the standard method (working on uncompressed text). Notice that the times are marginally larger than the 50% predicted because of the cost of filtering and compression (which are nearly 20% of the total time).

Files	Compressed Filter Method			Standard	Compressed /
	Flt.& Compr.	Sorting	Total	Method	Standard
AP	24	93	117	193	60.62%
DOE	18	54	72	110	65.45%
FR	20	72	92	171	53.80%
WSJ	26	98	124	223	55.60%
ZIFF	24	93	117	201	58.21%

Table 6. Indexing time (in elapsed minutes)

Table 7 presents the space relationship between (i) the compressed text plus index and the original text and (ii) the compressed text plus index and the original text plus index.

File	Of the Original Text	Of the Original Text + Index
AP	93.32%	56.36%
DOE	89.22%	54.84%
FR	89.71%	55.16%
WSJ	92.42%	56.00%
ZIFF	94.69%	57.25%

Table 7. Percentual of the compressed text plus index, using spaceless words Huffman

The central part of the querying cost is the disk seek time to access each position in the text. Although a binary search is performed over the suffix array,

the text has to be accessed to retrieve the strings to compare. This involves a seek to a random text position, a phenomenon that becomes even more problematic when the database is stored on optical disk. A random seek is known to traverse 1/3 of the text file, i.e. a constant fraction [BNB+95]. Since our compressed text takes 25-30% of the original one, the seek times (and hence querying times) are reduced in the same fraction, i.e. querying could be 3-4 times faster. However, as we see later, many factors reduce this improvement to being only twice as fast in practice.

Table 8 shows the query times for the compressed and uncompressed versions. Queries were words and phrases randomly selected from the text and filtered as previously explained. The suffix array and the text were kept in separate disks. We averaged 1,000 samples to obtain each data point (this process has an intrinsically large standard deviation, near 20%). The standard deviation of our estimator is below 0.6%, and therefore the results are within 2% of the real values with a confidence level of 99%. As is typical of suffix arrays, retrieval times are almost independent of the number of words in the query.

File	1 word	2 words	3 words	4 words
AP	0.484 (55.52%)	0.468 (54.73%)	0.454 (54.98%)	0.450 (54.25%)
DOE	0.324 (48.32%)	0.320 (47.35%)	0.300 (46.72%)	0.302 (47.80%)
FR	0.419 (54.46%)	0.416 (55.95%)	0.355 (50.30%)	0.361 (52.90%)
WSJ	0.549 (60.28%)	0.537 (58.57%)	0.517 (57.85%)	0.513 (56.34%)
ZIFF	0.526 (64.06%)	0.496 (63.18%)	0.484 (62.77%)	0.468 (62.25%)

Table 8. Plain search times (in seconds), and fraction of the time of the uncompressed version

As it can be seen, the time reduction is of 55-60%, smaller than expected. This is because although seeking in the text costs nearly 1/3 of the uncompressed version, the suffix array is not compressed and therefore accessing it costs the same in both cases. Additional search structures over the suffix array [BZ95, BYBZ96] improve the time reduction by a factor of more than 5 times, by performing the first steps of the binary search in main memory (where some pruned suffixes are stored), and reading into memory only one block of the suffix array. The binary search ends in this block. Therefore, the disk accesses are: one to read the suffix array block plus the random accesses to the text performed to search in that block.

To give an idea of the improvement obtained by using additional structures, we separate the times to access the text, which is shown in Table 9. Although the total query time is still worse than using additional structures (since we still perform *all* the text accesses), this should be a good approximation to the reduction between the compressed and the uncompressed version using additional structures. As seen, the times are reduced to nearly 60% than the plain version, but the times of the compressed version over the uncompressed one improve only to 40-50% instead of the expected 25-30%. This is due to many factors, such as the disk latency and transfer times (which do not improve by compressing

the file) or disk fragmentation, which prevents the contiguous text allocation on disk that would take full advantage of the reduced sizes.

File	1 word	2 words	3 words	4 words
AP	0.287 (45.57%)	0.274 (44.25%)	0.261 (43.63%)	0.255 (42.97%)
DOE	0.173 (35.56%)	0.169 (34.94%)	0.156 (33.54%)	0.157 (34.56%)
FR	0.240 (42.72%)	0.244 (44.85%)	0.195 (38.47%)	0.195 (40.18%)
WSJ	0.325 (48.66%)	0.313 (46.75%)	0.296 (45.15%)	0.291 (44.19%)
ZIFF	0.334 (56.10%)	0.309 (53.83%)	0.299 (53.48%)	0.284 (52.21%)

Table 9. Search times considering only text accesses (in seconds), and fraction of the time of the uncompressed version

5 Conclusions and Future Work

We presented a technique to index compressed text, oriented to suffix arrays. Our main contributions are: a new technique to treat blanks which is slightly superior to the standard one, the idea of a compression code that preserves the lexicographical ordering of the words (which allows to sort the strings without decompressing), and a suffix array index built directly on the compressed text. We also presented a new way of building a suffix array for plain uncompressed text that takes approximately half the time of the best known algorithm [GBYS92]. Finally, we have shown that query times are also reduced by half on compressed text. We explained the methods and studied their performance, both analytically and experimentally.

The main results are that we can handle the compressed text throughout the whole process with *improved* time and space performance. The text can be compressed to 25-30% of its original size, and the whole scheme (index plus text) takes 55-60% of the uncompressed version, and still less than the original text with no index. Moreover, indexing and querying are nearly twice as fast than in the uncompressed version of the index.

This scheme can be readily adapted to meet other requirements. For example, it is not difficult to mix it with the approach of [ANZ97] to allow searching for regular expressions, approximate patterns, etc. This is because that approach is mainly based on processing the vocabulary, which is stored in our index. We are currently working on this, as well as on a more complete system capable of compressing and indexing whole collections of documents. Another line of work we are pursuing is to handle modifications to a text database (insertions, deletions, updates). Finally, we can add search structures such as [BZ95] on top of the suffix array to improve the performance significantly (reductions by a factor of more than 5 times are reported).

We have not attempted to compress the suffix array itself. This is because the pointers stored in the array are known to be truly random, in the sense that they represent a random permutation of the index points of the text. Therefore,

the suffix array cannot be significantly compressed (e.g. if we use Gzip over the suffix array of AP or WSJ collections we get 95-96% of the original size).

Although there is already some work done on building an inverted list index on compressed text [MB95, ZM95], we believe that some of the ideas presented here can be applied there, for example the spaceless words model. The time improvements, on the other hand, are specific of suffix arrays and are not likely to be obtained on inverted lists. When we have added the mentioned search structures [BZ95] on top of our scheme, we plan to compare it against this compressed inverted lists scheme. Suffix arrays should be superior in searching phrases and complex queries, though the influence of compression in this known result has not been yet assessed.

References

[ABF96] A. Amir, G. Benson and M. Farach. Let sleeping files lie: pattern matching in z-compressed files. *Journal of Computer and Systems Sciences*, 52(2):299-307.

[ANZ97] M. D. Araújo, G. Navarro and N. Ziviani. Large text searching allowing errors. In R. Baeza-Yates, editor, *Proc. of WSP'97*.

[BMN+93] T. C. Bell, A. Moffat, C. Nevill-Manning, I. H. Witten and J. Zobel. Data compression in full-text retrieval systems. *Journal of the American Society for Information Science*, 44:508-531, 1993.

[BNB+95] E. Barbosa, G. Navarro, R. Baeza-Yates, C. Perleberg and N. Ziviani. Optimized binary search and text retrieval. In *Proc. of ESA'95*, 1995, pages 311–326. LNCS 979.

[BSTW86] J. Bentley, D. Sleator, R. Tarjan and V. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29: 320-330, 1986.

[BWC89] T. C. Bell, I. H. Witten. and J. G. Cleary. Modeling for text compression. *ACM Computing Surveys*, 21 (4): 557-591, 1989.

[BYBZ96] R. Baeza-Yates, E. F. Barbosa and N. Ziviani. Hierarchies of Indices for Text Searching. *Information Systems 21* (6): 497–514, 1996.

[BZ95] E. F. Barbosa and N. Ziviani. From Partial to Full Inverted Lists for Text Searching. *Proc. of WSP'95*, Valparaíso, Chile, 1995, pages 1–10.

[FBY92] W. B. Frakes and R. Baeza-Yates (Eds.). *Information Retrieval: Data Structures and Algorithms*, Prentice-Hall, Englewoods Cliff, N.J., 1992.

[FT95] M. Farach and M. Thorup. String matching in Lempel-Ziv compressed strings. *Proc. 27th ACM Annual Symposium on the Theory of Computing*, 1995, pages 703-712.

[GBYS92] G. H. Gonnet, R. Baeza-Yates and T. Snider. New indices for text: PAT trees and PAT arrays. In W. B. Frakes and R. Baeza-Yates (Eds.), *Information Retrieval: Data Structures and Algorithms*, 66-82, Prentice-Hall, Englewoods Cliff, N.J., 1992.

[Gon87] G. H. Gonnet. PAT 3.1: An Efficient Text Searching System. *User's Manual.* Center for the New Oxford English Dictionary. University of Waterloo, Waterloo, Canada, 1987.

[Har95] D. K. Harman. Overview of the Third Text REtrieval Conference. In *Proc. of the Third Text REtrieval Conference (TREC-3)*, pages 1-19, National Institute of Standards and Technology Special Publication 500-207, 1995.

[Hea78] J. Heaps. *Information Retrieval - Computational and Theoretical Aspects.* Academic Press, 1978.

[HT71] T. C. Hu and A. C. Tucker. Optimal computer-search trees and variable-length alphabetic codes. *SIAM J. Applied Math.*, 21: 514-532, 1971.

[Huf52] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. of the I.R.E.*, 40(9): 1090-1101, 1952.

[Knu73] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, Reading, Mass., 1973.

[Man93] U. Manber. A text compression scheme that allows fast searching directly in the compressed file. Technical Report 93-07. Dept. of Computer Science, University of Arizona, March 1993.

[MB95] A. Moffat and T. C. Bell. In situ generation of compressed inverted files. *Journal of the American Society for Information Science*, 46: 537-550, 1995.

[MM90] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searchs. *1st ACM-SIAM Symposium on Discrete Algorithms*, 319-327, San Francisco, 1990.

[Mof89] A. Moffat. Word-based text compression. *Software Practice and Experience*, 19(2):185-198, 1989.

[WBN92] I. H. Witten, T. C. Bell and R. Neal. Indexing and compressing full texts databases for CD-ROM. *Journal of Information Science*, 17: 265-271, 1992.

[WNC87] I. H. Witten, R. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 29(4): 320-330, 1986.

[Zip49] G. Zipf. *Human Behaviour and the Principle of Least Effort.* Addison-Wesley, 1949.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3): 337-343, 1977.

[ZL78] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5): 530-536, 1978.

[ZM95] J. Zobel and A. Moffat. Adding compression to a full-text retrieval system. *Software Practice and Experience*, 25(8): 891-903, 1995.

This article was processed using the LaTeX 2_ε macro package with CUP_CS class